# ChatDB – Taking to Database Management Systems using Natural Language
# Final Report

**Group 8 - Team Members:**
- Yu-Ching Huang (jhuang13@usc.edu)
- Shih-Hui Huang (shihhuih@usc.edu)
- Yu-Chen Lu (ylu74747@usc.edu)

## I. Introduction

In this project, we developed ChatDB — a system that allows users to interact with both SQL and NoSQL databases using natural language through LLM API (GPT-4) and an intuitive web-based interface.

ChatDB interprets user input and translates it into executable SQL or MongoDB queries, and supports schema exploration, data querying, and data modification operations:

1. **Schema Exploration:** Users can explore database schemas and data by identifying tables or collections, viewing their attributes, and retrieving sample data records.
2. **Query Execution:** ChatDB supports SQL queries such as SELECT, FROM, WHERE, GROUP BY, HAVING, ORDER BY, LIMIT, OFFSET, and multi-table JOINs. It also supports MongoDB queries including find with projections, and aggregate pipelines with $match, $group, $sort, $limit, $skip, $project, and $lookup.
3. **Data Modification:** ChatDB enables users to insert, update, and delete data seamlessly using natural language instructions.

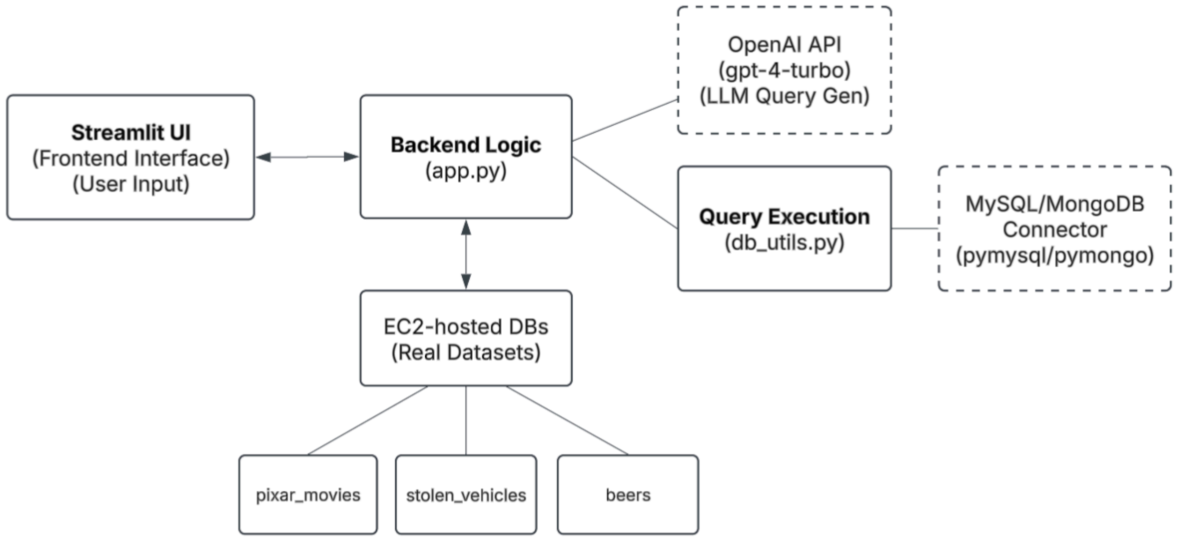
## II. Planned Implementation (From Project Proposal)

Our original plan was to implement Llama 2 locally to transform natural language into SQL queries. However, after comparing the results in terms of response accuracy, speed, and ease of integration, we decided to go with the OpenAI API. For SQL databases, we will use MySQL, and for NoSQL databases, we will utilize MongoDB.

The interface will support various functions in both SQL and NoSQL databases. Users would be able to explore database schemas and data by identifying tables or collections, viewing their attributes, and retrieving sample data. The ChatDB will be able to convert natural language into appropriate SQL or NoSQL commands to execute data retrieval operations, supporting SQL clauses like SELECT, WHERE, JOIN, GROUP BY, and functions in NoSQL databases such as find, aggregate, and $lookup. Additionally, the interface will handle data modification requests, enabling users to insert, update, and delete data seamlessly using natural language instructions.

In addition, we will develop a web-browser-based user interface that will provide an intuitive and user-friendly platform for users to interact with ChatDB, allowing them to view responses in a structured format.

## III. Architecture Design (Flow Diagram and its description)



The ChatDB system consists of four main components:

1. **Frontend Interface (Streamlit, app.py):** Users interact with the application via a simple Streamlit-based web interface. They can select a database type **(MySQL or MongoDB)**, pick a dataset **(pixar_movies, beers, or stolen_vehicles_db)**, and enter a natural language question. When they submit, the frontend sends this input to the backend for processing.

2. **Backend Logic:** The backend builds a prompt that includes the selected database type, dataset schema, and the user's natural language question. This prompt is sent to the OpenAI API (model gpt-4-turbo) to generate a query in either PyMySQL or PyMongo syntax, depending on the selected database type. The backend ensures the generated query follows strict formatting rules.
   - **LLM Service (OpenAI API, app.py)**: The OpenAI API processes the prompt and generates a database query that is syntactically and semantically appropriate. The response is then sent back to the backend for execution.
   - **Database Layer (MySQL and MongoDB hosted on EC2, db_utils.py):** Once the query is generated, the backend uses helper functions in db_utils.py to connect to the appropriate database. For MySQL, it uses pymysql; for MongoDB, it uses pymongo. The query is executed against the selected dataset hosted on an EC2 instance. Results (or errors) are returned as a Pandas DataFrame.

The results are rendered back to the user in an interactive DataFrame. If there's an error, it is caught and displayed with an appropriate message.

## IV. Implementation

1. **Functionalities**

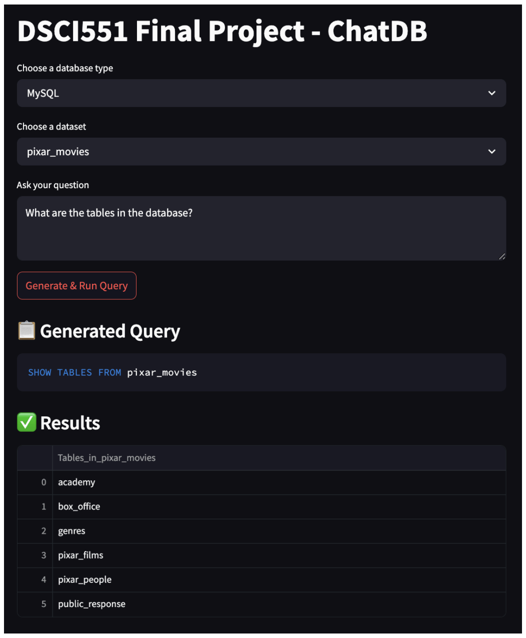
   - Natural Language Query Interface: Users can input plain English queries to retrieve or modify data from structured databases.

- Multi-Database Support: Supports both relational (MySQL) and non-relational (MongoDB) databases.
- Dataset Selection: Offers predefined datasets—pixar_movies, beers, and stolen_vehicles_db—each with multiple tables/collections.
- Schema-Aware Prompting: Automatically includes schema information in prompts to ensure accurate query generation.
- Real-Time Query Generation and Execution:
    - For MySQL: Translates input to SQL using PyMySQL syntax.
    - For MongoDB: Translates input to MongoDB commands using PyMongo.
- Dynamic Result Display: Renders query results as interactive tables within the Streamlit web app.
- Error Handling: Provides feedback for invalid queries or connection issues.

## 2. Tech Stack

- Frontend: Streamlit – For building an interactive web interface.
- Backend:
    - app.py – Manages UI logic, prompt construction, and LLM interaction.
        - LLM Service: OpenAI GPT-4 Turbo API – For transforming natural language into structured queries.
    - db_utils.py – Handles connection and execution for both SQL and NoSQL databases.
        - MySQL (v8.0.41) – Hosted on EC2, connected via PyMySQL.
        - MongoDB (v6.0) – Hosted on EC2, connected via PyMongo.
- Cloud Hosting: AWS EC2 – Hosts the MySQL and MongoDB instances.
- Dependencies: streamlit, openai, pymysql, pymongo, pandas

## 3. Implementation Screenshots - MySQL (Left), MongoDB (Right) using Pixar Movies


DSCI551 Final Project - ChatDB

Choose a database type: MySQL

Choose a dataset: pixar_movies

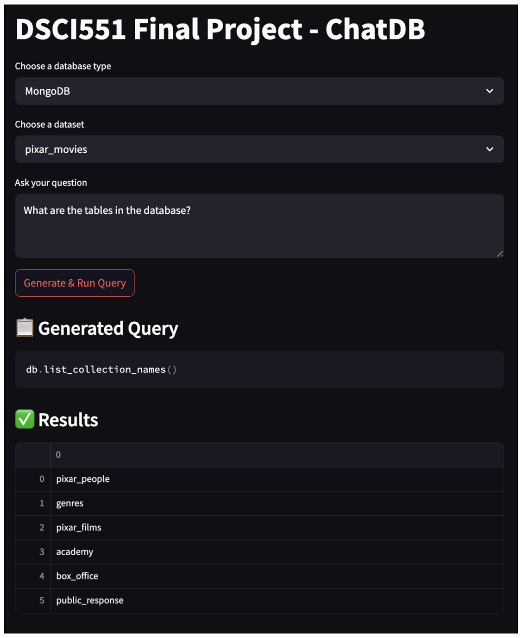
Ask your question: What are the tables in the database?

📋 Generated Query

```
SHOW TABLES FROM pixar_movies
```

✅ Results

| | Tables_in_pixar_movies |
|---|---|
| 0 | academy |
| 1 | box_office |
| 2 | genres |
| 3 | pixar_films |
| 4 | pixar_people |
| 5 | public_response |


DSCI551 Final Project - ChatDB

Choose a database type: MongoDB

Choose a dataset: pixar_movies

Ask your question: What are the tables in the database?

📋 Generated Query

```
db.list_collection_names()
```

✅ Results

| | 0 |
|---|---|
| 0 | pixar_people |
| 1 | genres |
| 2 | pixar_films |
| 3 | academy |
| 4 | box_office |
| 5 | public_response |

## V. Learning Outcome

Through this project, we gained valuable hands-on experience in applying database concepts and integrating LLM APIs with real-world datasets. Our key learnings include:

1. **Database Implementation with LLM API Integration**
    - Database Hosting & Management
        - Successfully deployed and managed both MySQL and MongoDB on AWS EC2 instances
        - Integrated real-world datasets and supported robust data querying and manipulation
    - Query Functionalities Implemented
        - MySQL (via PyMySQL): SELECT, FROM, WHERE, GROUP BY, HAVING, ORDER BY, LIMIT, OFFSET, JOIN
        - MongoDB (via PyMongo): $find, $match, $group, $sort, $limit, $skip, $project, $lookup
    - LLM Model Evaluation & Usage
        - Explored and compared multiple LLMs for database-related NLP tasks:
            - tiiuae/falcon-7b-instruct: Strong at natural language generation and handling complex instructions
            - codellama/CodeLlama-34b-Instruct-hf: Effective for Python code generation and translation; resource-intensive due to large model size
        - Leveraged LLM APIs to convert natural language queries into SQL and MongoDB operations, enabling dynamic interaction with databases
    - Streamlit Web Application
        - Built an interactive front-end interface allowing users to:
            - Input natural language queries
            - Execute translated queries in real time
            - Visualize results directly from MySQL or MongoDB
        - Designed for non-technical users to intuitively explore and extract insights from complex databases without writing code

2. **Optimized Prompt Engineering Strategies**

    Although OpenAI API did not require extremely detailed prompts, our experimentation with other APIs helped us learn how to refine prompt strategies to improve accuracy.
    - Provided comprehensive dataset schema information
    - Designed diverse functional examples to support a wide range of queries, including example natural language and corresponding query
    - Established strict formatting rules to ensure SQL output compatible with PyMySQL and valid PyMongo syntax
    - Added fallback mechanisms for manual debugging when the model generated invalid queries
    - Tailored prompts based on the specific database (SQL/NoSQL) and required query types

3. **Team Collaboration and Project Management**

   Working as a team, we developed stronger coordination and communication skills. For example, we improved task division, timeline management, code integration, and collaborative debugging—learning to approach issues from multiple perspectives and find solutions together.

## VI. Challenges Faced

1. **Hugging Face Model Size Limitations**

   In the early stages of our project, we experimented with several LLM APIs, including Hugging Face, Llama 2, and Ollama. However, we encountered several limitations. With Hugging Face, we initially aimed to use more powerful models for code generation and complex instruction comprehension, such as "tiiuae/falcon-7b-instruct", "mistralai/Mistral-7B-Instruct-v0.1", "google/flan-t5-xl", and "codellama/CodeLlama-34b-Instruct-hf". However, these models exceeded the resource constraints imposed by Hugging Face's free-tier API. Specifically, the model size exceeded the 10GB limit of free-tier inference endpoints. As a result, we switched to smaller models like "flan-t5-large" on Hugging Face, and also began exploring Ollama with locally hosted models like "mistral:7b-instruct", which did not impose similar usage limits on model size.

2. **Significantly Reduced in Performance with Smaller Hugging Face and Ollama Models:**

   We observed a significant drop in performance when using smaller models through Hugging Face or Ollama. The lightweight models often struggled to comprehend complex instructions and contextual nuances, which negatively impacted the quality of generated outputs, particularly for tasks involving reasoning, inference, or multi-step execution. Moreover, while Hugging Face offers a wide array of pre-trained and fine-tuned models optimized for different use cases, Ollama lacks the same level of ecosystem support. We attempted to mitigate this through prompt engineering, spending significant time on crafting detailed and structured prompts to make up for the models' limited understanding capabilities.

3. **Prompt Fine-Tuning Required & Slow Runtime**

   We had to write highly detailed prompts to improve the accuracy of generated outputs, often including comprehensive descriptions of our dataset schema and multiple concrete examples. For instance, for each query-related functionality, such as find with projections, aggregation operations like group by, sort, limit, multi-table joins, and data modification queries, we provided natural language inputs alongside corresponding query translation examples, as well as precise formatting rules.

   Nevertheless, we continued to encounter notable errors, especially in MongoDB queries compared to SQL. Additionally, some Hugging Face models had strict prompt token limitations for free-tier models, which created a dilemma: we needed detailed instructions to improve performance, but the models could not accept sufficiently long prompts. While Ollama supported more flexible prompt lengths and we implemented fallback mechanisms for manual debugging, and it eventually produced correct results, but the runtime was impractically slow, each natural language query translation could take up to five minutes.

4. **Final Decision**

   Ultimately, we decided to subscribe to the OpenAI API, which offers faster responses and better natural language understanding and reasoning. The accuracy of generated SQL and MongoDB queries was substantially higher than what we achieved with Hugging Face or Ollama, even without the need for overly complex prompt engineering. Additionally, it delivered reliable results with minimal guidance, which streamlined our workflow and significantly improved overall system performance. Consequently, we selected OpenAI as the final LLM API for our project implementation.

## VII. Individual Contribution

Our project was carried out as a group effort, and we met frequently to brainstorm ideas, divide tasks, and work through implementation challenges together. Most of the development process, including coding, testing, and debugging, was done collaboratively.

One area where we initially worked independently was during the selection of a suitable API. Each member explored a different option to evaluate performance and feasibility: **Yu-Ching** tested the OpenAI API, **Shih-Hui** Huang experimented with Ollama, and **Yu-Chen** tried the Hugging Face API. After comparing the results in terms of response accuracy, speed, and ease of integration, we concluded that the OpenAI API best met our needs and decided to adopt it for the project. From that point onward, we continued working as a team to complete the remaining components and refine the overall system.

## VIII. Conclusion

ChatDB successfully bridges natural language input with SQL and NoSQL databases, enabling users to explore and manipulate data without needing database query expertise. By leveraging OpenAI API, we achieved accurate and efficient translation of natural language into MySQL and MongoDB queries, supporting a wide range of operations including schema exploration, complex data retrieval, and data modification.

While we initially explored open-source LLMs (Hugging Face and Ollama), limitations in performance, speed, and model size led us to adopt OpenAI API, which delivered significantly better accuracy and usability with minimal prompt tuning. Through this project, we gained hands-on experience with prompt engineering, full-stack development using Streamlit, database deployment on AWS EC2, and practical integration of LLM APIs.

Despite challenges such as inconsistent MongoDB outputs and managing model limitations, ChatDB met its goals of accessibility, flexibility, and real-time interaction. This project not only demonstrates the power of LLMs in data systems but also lays the groundwork for future enhancements like conversation memory, support for additional databases, or domain-specific fine-tuning.

ChatDB not only supports a wide range of query and data modification operations but also provides a strong foundation for future work in natural language database interfaces. This project has equipped us with deep, hands-on experience in database systems, LLM integration, cloud deployment, and full-stack development — key competencies that are increasingly vital in the evolving field of AI-driven data access.

## IX. Future Scope

As ChatDB successfully achieves its core goal of enabling natural language interaction with SQL and NoSQL databases, there are several promising directions for future development. These enhancements aim to make the system more conversational, customizable, and applicable across diverse user needs and domains.

1. **Conversation Memory (Multi-Turn Interaction)**

   Currently, ChatDB handles each query independently without retaining prior context. Adding conversation memory would enable multi-turn interactions where the system can remember previous queries, follow-up questions, or clarified constraints.

   For example, a user could first ask “Show me all films released after 2010,” then follow up with “Now sort those by rating.” With conversation memory, ChatDB could retain the first query’s context and apply the new instruction without needing a full restatement. This would make the interface more natural and closer to how humans interact with data.

2. **Support for Additional Databases (User-Uploaded Data)**

   In the current implementation, ChatDB supports fixed datasets hosted on MySQL and MongoDB. As a next step, we aim to allow users to upload their own datasets, supporting formats such as CSV, JSON, and SQL dump files — and automatically integrate these into a temporary or user-specific database instance. Once uploaded, ChatDB would dynamically parse the schema, update the LLM prompts, and allow users to query their own data via natural language. This would significantly expand ChatDB’s flexibility and usability across industries and use cases.

3. **Domain-Specific Fine-Tuning**

   While OpenAI API performs well out of the box, its general-purpose nature means it may occasionally misunderstand queries that involve specialized terminology or industry-specific logic (e.g., finance, healthcare, retail). Future versions of ChatDB could include fine-tuned LLMs trained on domain-specific query pairs — natural language instructions and their correct SQL or NoSQL equivalents for particular datasets. This would improve the accuracy of query generation and make ChatDB a more powerful tool in professional environments where precision is critical.

## Appendix

**Links:** Google Drive | Github

**Datasets Used Downloaded from**
https://mavenanalytics.io/data-playground?dataStructure=Multiple%20tables&order=date_added%2Cdesc&page=1&pageSize=5

**1. Pixar Films Dataset: (Pixar Films)**
Eric Leung created and maintains an R package and dataset that chronicles the full history of Pixar films — from Toy Story in 1995 to Inside Out 2 in 2024. This resource offers rich, structured data on each movie, including information on the creative team (writers, directors, composers, producers), financial details like budget and box office revenue, critic ratings, and recognition at the Oscars.

**2. Stolen Vehicles Dataset: (Motor Vehicle Thefts)**
This dataset contains six months of records from the New Zealand Police's Vehicle of Interest database, documenting individual stolen vehicles. Each entry includes details such as vehicle type, make, model year, color, date of theft, and the region where the theft occurred.

**3. Beers:**
We used the beers dataset used during in-class exercise.